ENSF 594 Assignment 2 report:

Zach Frena

1. The data collected (use tables and graphs to help illustrate this):

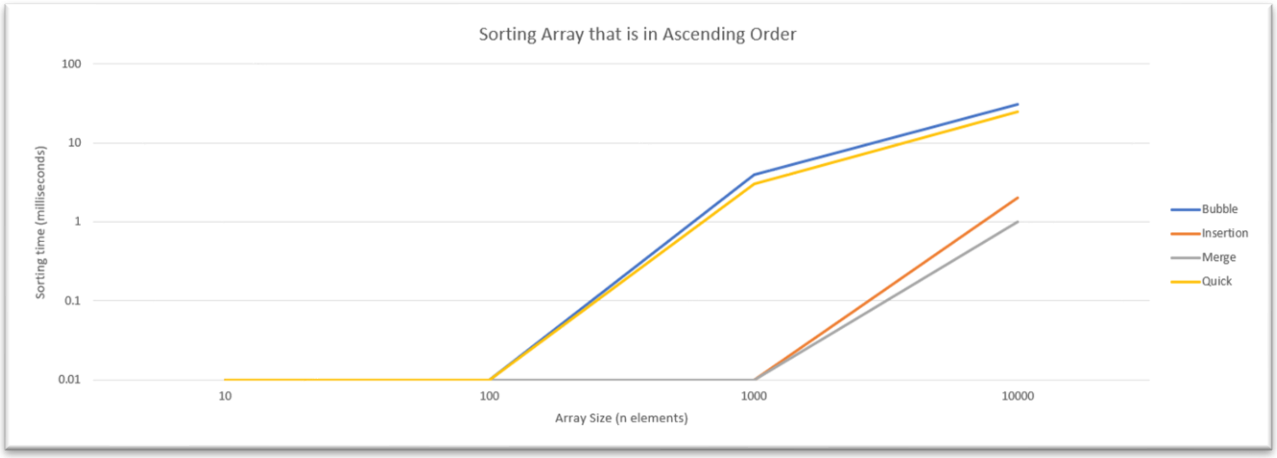
	Sorting Time (milliseconds)											
	Bubble			Insertion			Merge			Quick		
Array size	Ascending	Random	Descending	Ascending	Random	Descending	Ascending	Random	Descending	Ascending	Random	Descending
10	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
100	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
1000	4	2	3	0.01	1	0.01	0.01	0.01	0.01	3	1	1
10000	31	127	62	2	31	18	1	1	1	25	2	43
100000	1224	14315	6173	3	896	1841	6	17	9	StackOverFlowError	12	StackOverFlowError
1000000	126781	1705507	611507	5	298399	210669	63	156	72	StackOverFlowError	101	StackOverFlowError

Notes:

i) In the above table, any cell that has the value of "0.01 milliseconds" was actually calculated/recorded as "0 milliseconds" (which is still >0, just very small) by my program. This has been done to allow the y-axis to be plotted on a logarithmic scale in excel, which requires all numerical values > 0.

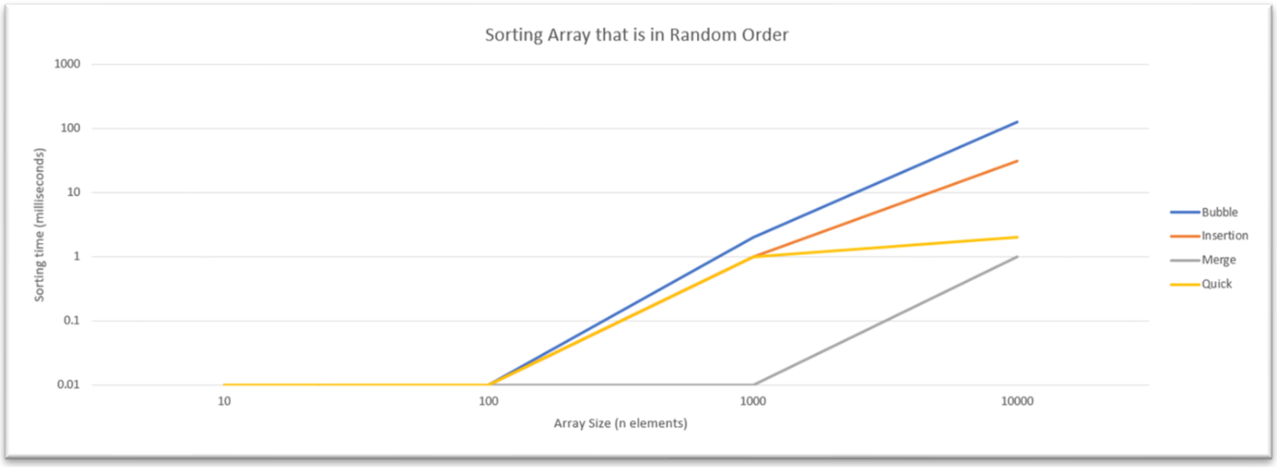
ii) For the array sizes 100,000 and 1,000,000, the quick sort algorithm was unable to process the ascending and descending ordered arrays without throwing a *StackOverFlowError*. As specified in the rubric, I have only plotted the data from the arrays sized 10 to 10,000.

Best Case Scenario (Array is in Ascending Order):



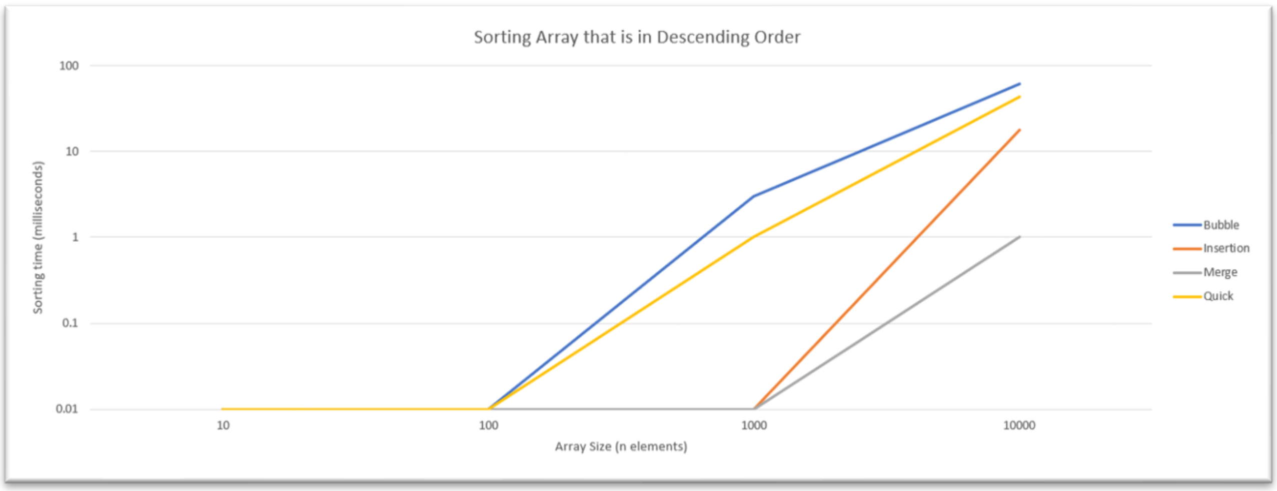
As seen in the graph above, the times required to sort 10, 100, and even 1000 items in our ascending array are very small, and the time needed to sort 10,000 items never exceeds 50 milliseconds. This makes sense because the array is already in the proper order and only a small number of computational exercises (swaps, insertions, etc.) needs to be performed.

Average Case Scenario (Array is in Random Order):



In the above graph, we see the performance of the sorting algorithms in the average case (randomly ordered array). There is a large difference in the magnitudes of sorting time between the merge & quick sort algorithms and the bubble & insertion sort algorithms. This makes sense because the average-case time complexity is $O(n \log n)$ for both quick and merge sort, whereas the average-case time complexity for bubble and insertion sort is $O(n^2)$. Therefore, we expect the bubble and insertion sorts to take longer.

Worst Case Scenario (Array is in Descending Order):



The graph above shows the performance of the various sorting algorithms in the worst-case scenario (i.e. the array is in descending order). The bubble, insertion, and quick sorting algorithms have much higher sorting times than merge sort. This makes sense because the time complexity for the worst-case is $O(n \log n)$ for merge sort, whereas the worst-case time complexity for bubble, insertion, and quick sort is $O(n^2)$.

2. Data analysis. What does the data tell us about the algorithms?

The data tells us 2 things: Firstly, that certain algorithms are more suitable for larger sets of data than others, and secondly, that the order of the input array **does** also matter in how performative the sorting algorithms are. The data that was collected for very large arrays (100,000 and 1,000,000) for the bubble, insertion, and merge sorting algorithms also shows us that the sorting time increases very rapidly on a non-linear basis as the array size becomes very large.

3. The complexity analysis of each algorithm (retrieved from https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/ on 07/15/2021)

Algorithm	Best Case (Ascending order)	Average Case (Random order)	Worst Case (Descending order)
Bubble	n	n^2	n^2
Insertion	n	n^2	n^2
Merge	$n \log n$	$n \log n$	$n \log n$
Quick	$n \log n$	$n \log n$	n^2

4. Interpretation. What does the empirical data and complexity analysis tell us about the algorithms? How do the algorithms compare with each other? Does the order of input matter? How do algorithms within the same big-O classification compare to each other?

The empirical data gathered from the outputs of the program support the theoretical complexity analysis of the algorithms:

1) In the best-case scenario (when the input array is already sorted): bubble sort and merge sort have a small complexity advantage over merge and quick sort. However they are all very close in time-complexity and have a high performance, even when the array size approaches large values like 10,000.
2) In the average-case scenario with a randomized input array: both the bubble and insertion sort algorithms have a quadratic time-complexity which makes them slower than the merge and quick sort algorithms. This is shown in the empirical data and becomes increasingly clearer as the array size increases.
3) In the worst-case scenario (input array is descending): bubble, insertion, and quick sort algorithms all have a quadratic time-complexity. As seen in the empirical data, merge sort has a much lower sorting time because it just has an $O(n \log n)$ time-complexity.

Amongst the algorithms that have the same big-O classification, both bubble and insertion sort have the same big-O(), however the insertion sort is faster due to the mechanism of the sorting.

5. Conclusions. What algorithms are appropriate for practical use when sorting either small or large amounts of data? Are there any limitations with any of the algorithms? How does your analysis support these conclusions?

For smaller arrays (size < 1000), there does not appear to be a large importance on the type of sorting algorithm you choose to employ. Many people prefer to use insertion sort for small arrays due to its simplicity. However, for larger arrays it is important to choose a sorting algorithm that does not have a time-complexity that grows exponentially or cubically with N. For this reason (and with empirical data that supports this conclusion), larger arrays should be sorted with either quick sort or merge sort. Even in the average case (which we expect), the bubble and insertion sorting algorithms have time-complexities of n^2, which will greatly impact processing time when the array gets very large. One limitation that quick sort (naively implemented) has when processing large arrays (>100,000) is that it often results in a *StackOverFlowError* when trying to sort the ascending or descending order array. A solution to this is to shuffle the array to break up the order to gather the "best-case" and "worst-case" scenario.

Citations:

1. Baeldung, "Merge Sort in Java," *Baeldung*, 22-Sep-2020. [Online]. Available: https://www.baeldung.com/java-merge-sort. [Accessed: 15-Jul-2021].

2."Bubble Sort," *GeeksforGeeks*, 28-Jun-2021. [Online]. Available: https://www.geeksforgeeks.org/bubble-sort/. [Accessed: 15-Jul-2021].

3."Insertion Sort," *GeeksforGeeks*, 08-Jul-2021. [Online]. Available: https://www.geeksforgeeks.org/insertion-sort/. [Accessed: 15-Jul-2021].

4."QuickSort," *GeeksforGeeks*, 28-Jun-2021. [Online]. Available: https://www.geeksforgeeks.org/quick-sort/. [Accessed: 15-Jul-2021].

5."Time Complexities of all Sorting Algorithms," *GeeksforGeeks*, 28-Jun-2021. [Online]. Available: https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/. [Accessed: 15-Jul-2021]